APPLICATION FOR UNITED STATES LETTERS PATENT

SCALABILITY, AVAILABILITY, AND MANAGEMENT FEATURES FOR BUSINESS COMMERCE SERVER

Inventors:

Krishna Kanth
3535 Shafer Drive
Santa Clara, CA 95051
A Citizen of India

Hanuma Kodavalla
32415 Monterey Drive
Union City, CA 94587
A Citizen of India

Hooman Moobed
557 Edelweiss Drive
San Jose, CA 95136
A Citizen of U.S.A.

Ravi Sharma
34220 O'Neil Terrace
Fremont, CA 94555
A Citizen of India

Siamak Darakhshan
15238 Cooper Avenue
San Jose, CA 95124
A Citizen of India

Assignee:
Asera, Inc.
600 Clipper Drive, Suite 100
Belmont, CA 94002
A Delaware corporation

Entity: Small

Beyer, Weaver & Thomas LLP
P.O. Box 778
Berkeley, CA 94704
Tel: (650) 961–8300

Attorney's Docket No. ASRAP012

SCALABILITY, AVAILABILITY, AND MANAGEMENT FEATURES FOR BUSINESS COMMERCE SERVER

RELATION TO PRIOR APPLICATIONS

This application is related to the following -- U.S. Provisional patent application having serial number 60/164,021 (Attorney Ref. No. ASRAP001P), entitled “Method and Apparatus to Provide Custom Configurable Business Applications From a Standardized Set of Components,” filed August 23, 1999; Utility patent application having serial number 09/440,326 (Attorney Ref. No. ASRAP001), entitled "Method for Providing Custom Configurable Business Applications from a Standardized Set of Components," filed November 15, 1999; Utility patent application having serial number 09/439,764 (Attorney Ref. No. ASRAP002), entitled "Apparatus to Provide Custom Configurable Business Applications from a Standardized Set of Components," filed November 15, 1999; Utility patent application having serial no. 09/658,415 (Attorney Ref. No. ASRAP003, entitled "Method For Developing Custom Configurable Business Applications," filed September 8, 2000; and Utility patent application having serial no. 09/658,416 (Attorney Ref. No. ASRAP014), entitled " Integrated Design Environment For A Commerce Server System," filed September 8, 2000; Utility patent application having serial no. ________ (Attorney Ref. No. ASRAP005), entitled "Method for Providing Template Applications For Use By a Plurality of Modules," filed October 25, 2000; Utility patent application having serial no. ________ (Attorney Ref. No. ASRAP006), entitled "Method and Apparatus for Providing News Client and Server Architecture and Protocols," filed October 17, 2000; Utility patent application having serial no. 09/684,491 (Attorney Ref. No. ASRAP008), entitled "Adapter And Connector Framework For Commerce Server System," filed October 4, 2000; Utility patent application having serial no. ________ (Attorney Ref. No. ASRAP009), entitled "E-Commerce Application Built Using Workflows On A Workflow Engine And Methods Thereof," filed on the same date herewith; Utility patent application having serial no. ________ (Attorney Ref. No. ASRAP010), entitled "Presentation Layer For Business Application Development And Methods Thereof," filed on the same date herewith; -- each of which are hereby incorporated by reference in their entirety.

FIELD OF THE INVENTION

The present invention relates generally to scalability, availability, and management features for a server running customized business applications. In particular the features provide a more fault tolerant system with transparent fail-over to backup systems or components.

BACKGROUND OF THE INVENTION

The prior referenced applications provide for methods and apparatuses for creating custom configurable business or channel applications from a standardized set of components. More specifically, the referenced invention allows each business to select from a set of applications, customize that set of applications, and/or develop new customized applications from a set of development components. The prior applications provide for a server based method wherein best-of-breed services and/or applications are integrated in a seamless fashion and offered to enterprise businesses which develop customized business service applications through using the system. The server device is previously (and hereafter) referred to as the Asera Commerce Server (ACS).

The ACS includes a Commerce Server that provides a core set of technology (or application) services. A unique architecture and framework are provided by the Commerce Server, which facilitates development and use of customized applications. Most interactions with external systems or users are managed as business objects. The service application code is maintained separate from the data. This enables the system to quickly include (and/or change) new business processes or technology components without having to write substantial amounts of new code. The business result is more rapid customer deployments and/or modifications that are customized to include (if desired) the proprietary or competitive business practices of a contracting company.

The ACS can be viewed as a form of ASP (Application Service Provider). An ASP is generally an outsourcing business model. The ASP business model requires an open and extendable architecture that allows a system to implement a customer specific business solution in a short period of time. The ACS takes best-of-breed applications and incorporates them into one integrated solution to provide the ASPs. The architecture is scalable and extensible. A customized business (or channel) application solution is built for each enterprise company. The solution uses a "modular" or step-wise or "plug and play" approach towards building new applications. An enterprise company can then quickly acquire a turn-key e-commerce solution to automate their channel relationships. The system presents little (or no) risk for the enterprise company because a solution is built by the present system. The costs of undertaking such a development exist as a fixed development cost of the system. Any resulting customized solutions are implemented in considerably less time than previous systems. The enterprise company might pay for the application services on a cost per transaction, or a fixed fee basis.

The ACS is used to capture the particularized (or specific) business processes for a given customer, and these business processes are converted into a set of customized applications. The ACS uses business steps and rules to construct the application. The objects are data representations. The steps are business operations with a defined set of input and output ports, with each port also having a defined set of parameters. The business rules are used to capture customer specific business practices. A unique tool that employs a graphical user interface (GUI), allows a developer to arrange various steps (or composite steps) into business processes, or workflows. The tool provides library catalogs of steps to be applied to the various objects. The connections between steps are also verified as correct. A graphical display of the business process is shown, and rules can thereafter be applied to provide further customization by conditionally tagging certain points. Hence, to create a business process (or application) for any given business, tools are provided which allow modules (or steps) to be plugged or dropped into the potential process. The steps can be moved, or the connections modified. An initial person-to-person (or other type of) interview with the business (or customer) can be used to produce the framework for arranging the steps according to the needs of that particular business (i.e. customized routines). The modular aspect of the present system allows this to be done -- and modifications made -- in a relatively quick fashion. For instance, if a process has been created, but the customer wants it to behave in two different manners, then certain rules can be applied to provide the desired results, depending on conditional triggers that can be associated with the underlying business objects.

In general, Internet transactions can be divided into two categories: 1) business-to-business transactions, and 2) business-to-consumer transactions. Most solutions to automate transactions have dealt with business-to-consumer interactions. As such, these interactions are much more straightforward than business-to-business transactions. In a business-to-consumer transaction, the merchant supplies a "storefront" or web site that offers products to any number of diversified consumers who might wish to view this web site. The consumer then purchases a product via a selection and payment method, and the product is thereafter shipped to the consumer. On the other hand, when one business deals with another business, there is a much greater amount of business processes and customization in the transactions that occur. The ACS system therefore provides a customized business application that runs on the server and communicates with outside data sources.

As the ACS system continues to grow, there will be increasing focus on the performance of the service, the continuous availability of the service, and the ease of management. Often performance of a system is specified in terms of "adequate performance." This proves to be an ill defined phrase, primarily because the market for business-to-business electronic commerce systems and services is still immature. Given this, it is important for the ACS system to be able to grow from relatively small systems to very large systems as customer demands and needs grow.

Availability of the system is another important operational aspect of a successful business-to-business electronic commerce service. Customer perception of the service -- and more importantly revenue generation -- is directly dependent on availability. For some systems, it is acceptable for the service to be offline during peak hours for maintenance. However, as the service grows and becomes more global in nature, continual availability will become mandatory. With faults being inevitable in any large system, the system should have the ability to tolerate isolated faults in system components, and provide for transparent fail-over to backup systems and components until the fault is repaired.

Additionally, the administration of a large system is a complex task. Efficiency of administration correlates directly with the simplicity of system design and the existence of simple mechanisms to manage the complexity. Manageability of the system is also an evolving requirement that changes as the commerce server evolves. It then becomes important to provide simple mechanisms for starting and shutting down either the entire server, or its individual components, wherein individual components can be either a node in a cluster, or an application on a given node.

Therefore, what is needed are certain features for providing scalability, availability, and management features for the existing commerce server system. In particular, if a primary server fails, the application process should fail-over to the backup server in a transparent and efficient manner. For instance, the application process should be reconstructed to continue on the backup server without any error messages, or undue delay.

SUMMARY OF THE INVENTION

The present invention provides certain business objectives relating to scalability, availability, and manageability of the overall ACS system.

Scalability provides the ability to both scale-up on symmetric multi-processing (SMP) machines and the ability to scale-out on clusters of machines. Scale-up and scale-out both pertain to the ability of the system to proportionally increase the amount of work performed in a fixed quantum of time, as the hardware resources available for performing the work increases. Examples of work performed include the number of pages serviced by a web server, or the number of transactions processed by a database management system. Examples of hardware resources are the number of CPUs available, or the number of nodes in a cluster.

For scale-up on SMP machines, one objective is to remove as many of the synchronization bottlenecks as possible so that each additional processor added to the machine results in improved performance. For scale-out on cluster nodes, there are some inherent bottlenecks in existing Java virtual machines that make scaling up on SMP machines with more than 4 processors very challenging. Also, 2-4 processor SMP machines have become commodity items with drastically lower price points compared to 8, 16, or 32 processor SMP machines. Because of this price drop, four 4-processor SMP machines are generally cheaper than a single 16-processor SMP machine. It therefore becomes an attractive option to add the ability to cluster the cheaper commodity machines for increased performance.

Availability relates to the ability to transparently fail-over user sessions on a given cluster node to a backup cluster if that node fails. Each object pertaining to an application will be given a logical reference. Each application will have a state defined and thereby associated with the application. The various steps associated with an application, i.e. Composite Steps, Simple Steps, Interactive Steps, and Applications Steps, will be arranged in localized pool areas of a global pool. For each new update of a state, the associated logical references will be retrieved and serialized into a session object that gets sent to the backup server. From this minimized set of data, the backup server can reconstruct the application and continue to process it according to the existing state. This will provide a transparent fail-over to the backup server, without the overall need for error messages to be sent to the user. The user will also be able to proceed without the need for restarts and/or long processing delays.

Management (or manageability) provides the ability to add or drop nodes from the cluster without shutting down the service. Manageability also relates to the ability to easily deploy applications across all nodes of a cluster. A single system image is generally presented to the end-users. User experience remains the same irrespective of the number of nodes running the service in the cluster.

According to one aspect of the present invention, a method is provided for transparent backup service from a primary node to a backup node for applications having objects running on the primary node, the method comprising: defining a state for each application, the state having state parameters, and whereby the state can change with each user interaction with the application; determining pooling arrangements for storing the states associated with each application in a session object associated with a particular session; serializing the pooled states in the session object, with canonical representations being used for the various state parameters when available, and raw state parameter data being used otherwise; and periodically replicating the session object over to the backup node, wherein upon fail-over the backup node can be instructed to parse through the serialized data and reconstruct the application objects according to the given state at fail-over, thereby providing a transparent backup for running the application.

According to another aspect of the present invention, a distributed network server arrangement is provided for transparent backup service from a primary node to a backup node for applications having objects running on the primary node, the server arrangement comprising: a web server device for distributing requests from, and responses to, at least one browser device; at least one active primary node for running the applications; at least one backup node for running the applications in the event of fail-over of the primary node; and a separate session object associated with each browser device, each session object being used for storing the serialized state representations of the applications with canonical representations being used as available for state representation data, wherein the session object is periodically replicated over from the active primary node to the backup node, and the backup node can use the session object to reconstruct and run the application according to its state upon fail-over.

These and other aspects and advantages of the present invention will become apparent upon reading the following detailed descriptions and studying the various figures of the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention, together with further advantages thereof, may best be understood by reference to the following description taken in conjunction with the accompanying drawings in which:

Figure 1 shows a block diagram of a prior art main and backup server configuration.

Figure 2 shows, according to one aspect of the present invention, a block diagram of main and backup server configuration.

Figure 3 shows, according to one aspect of the present invention, a block diagram of a cluster configuration of representative components.

Figure 4 shows, according to one aspect of the present invention, a block diagram of a cluster configuration of representative components.

Figure 5 shows, according to one aspect of the present invention, a block diagram of primary and backup node configuration with the session object periodically sent over to the backup node.

Figure 6 shows, according to one aspect of the present invention, a block diagram of a Publish/Subscribe API as utilized to post cache invalidation messages from one node in the cluster to all of the rest.

Figure 7A shows, according to one aspect of the present invention, a block diagram wherein each application is assigned a defined state.

Figure 7B shows, according to one aspect of the present invention, a block diagram wherein the state is defined from certain necessary items needed to perform the application.

Figure 8 shows, according to one aspect of the present invention, a block diagram of the pools associated with the session object.

Figure 9 shows, according to one aspect of the present invention, a block diagram of the information flow for an example (keyword search) application.

Figure 10 shows, according to one aspect of the present invention, a block diagram of a composite step and certain associated representative steps.

Figure 11 shows, according to one aspect of the present invention, a block diagram of different step types and certain associated representative steps.

DETAILED DESCRIPTION OF THE INVENTION

The present invention provides for system configuration features, including aspects of scalability, availability, and manageability. In particular, redundancy is provided through the availability features. As applications steps are executed, in association with composite steps, single steps, and interactive steps, the states of the various applications are tracked, and sent over to a backup device. The state information is configured so that only an optimum amount needs to be sent over upon failure, and the backup device can transparently reconstruct the application, essentially where it left off on the primary device. The present mechanism will therefore keep track of the state, and then replicate the state over to the backup device, without significant overhead being added.

In prior systems, switching over to a backup device has generally caused a disruption of service between the user and the main server. Most prior systems do not provide for transparent fail-over. If the primary server fails, a message might be sent back to the user, and the user might be required to re-execute the transaction on the backup server. Note that some servers might automatically reconnect to a backup device, but they do not have any approach for re-executing all of the transactions that have been completed by a user up to the point of fail-over. Referring to Figure 1, a block diagram is shown of a representative prior art configuration. A main server 102 is shown handling the primary set of tasks to be performed. Upon failure of the main server 102, a backup server 104 is employed. The fail-over situation will require the user to re-execute the application on the backup server, which can be cumbersome and time consuming. Moreover, the user may be confronted with a series of error messages 106. These error messages will typically include instructions for the user to retry the transaction (once the backup system is employed). This will further disrupt the overall flow of performing the particular task or application.

Figure 2 next shows a block diagram of a representative configuration of the present system. The main server 202 is shown handling the primary set of tasks to be performed. The main server 202 stores logical references as keys (or other such data) 206 in storage area, database, or the like. Upon fail-over 208, the backup server 204 can use the keys (data) 206 to reconstruct the application, so that it will now be running on the backup server 204.

Figure 3 shows a representative block diagram 300 illustrating further details of node configuration. Requests from a browser 302 (or the like) enter the system through the load balancer device 304. The load balancer directs incoming requests to one of the configured web servers 306 and 308. Each web server in turn checks with the installed plug-ins to security applications 320 to see if the URI (Uniform Resource Identifier) is a protected resource. If yes, the plug-in authenticates the requesting user by challenging him or her for a valid user name and password (or similar entries).

Once the user is authenticated, URIs matching a specific pattern are redirected to plug-ins matching the application server(s) 310 and 312. The application server plug-ins are configured to communicate with one or more application servers. One of the servers to service the request is selected based upon a load balancing algorithm, with round-robin load balancing being the most commonly supported.

The ACS system is registered as a servlet within the Application server. Whenever the incoming request URI matches a specific pattern, the ACS is invoked via the appropriate servlet class. From thereon, execution proceeds between the user and the ACS.

Note that certain application server devices provide a mechanism for "startup" and "shutdown" classes. These classes are invoked whenever the application server is started or shutdown respectively. The ACS Application Server registers a startup class (i.e. AppServerStartup), which starts the partition daemon. A shutdown class is also registered (i.e. AppServerShutdown), which brings down the partition daemon when the application server is being shut down.

When a client (user) first requests the URI for this configuration, the application server plug-in directs the request to one of the application servers in the cluster and establishes a session for that client. The application server that services this request becomes the primary server for this session. The primary server then chooses a secondary server. In a 2-node cluster, the second node obviously acts as the secondary server. All subsequent requests within the same session are automatically redirected by the plug-in to the same primary server.

All servers in the cluster along with the application server plug-ins exchange "heart beats" at regular intervals. If the primary server fails for any reason, the application server plug-in detects the failure by missing heartbeats from that server. Any subsequent requests to sessions owned by the primary server are automatically redirected to the secondary server. This secondary server now becomes the primary server for those sessions. If there are more than two nodes in the cluster, then this server chooses one of the remaining servers in the cluster as the new secondary.

Requests can be configured to enter the ACS Application Server through a servlet, in this instance called AppServerServlet. This servlet can be configured to implement a generic interface. This is the generic application server interface that bridges any application server (i.e., Weblogic Server, Netscape Application Server, and so forth) and the ACS Application Server. For each new request, a new activity manager is instantiated, and control is passed to its execution method. Note also that the database 322 and/or shared disk 324 can be used to store the key (or logical reference) information for reconstructing a transactional flow upon failure of a primary server.

Cluster Installations are further described in association with the representative block diagram 400 of Figure 4. Cluster installations will generally have 3-tiers. Similar software components will be installed on each tier. The web server tier can have one or more web servers configured, as shown by web server 404 and 406. Configuring more than one web server provides better scalability for static content. It also provides higher availability in case one of the web servers fails. If more than one web server is configured, than a load balancer 408 should be included in the system. The load balancer directs incoming requests to one of the web servers, usually using the known round-robin load-balancing algorithms, or the like.

A cluster configuration also has two or more application servers 410 and 412 in the application server tier 414. Single node systems might be configured, but do not generally provide for the backup capabilities further described in the present system. In general, each of the application servers in the tier 414 should run the same logical routines and libraries. Such commonality might be furthered through a shared disk drive, or database server 416, in order to easily configure all of the servers and keep the software in sync.

Scalability. Scalability, in general, should not be construed as a functional change in the ACS Application Server. Functionality provided by the server will not change due to scalability improvements. The most noticeable visible change in the user experience will be improved performance of the commerce server and executing applications. Improving scalability translates to minimizing the waiting period spent in synchronized access to shared resources. Alternatively, scalability translates to minimizing shared resources. Example of shared resources on SMP systems are CPUs, network bandwidth, and available memory in the system. In the case of clusters, shared resources are typically data accessed by member nodes.

Scaling bottlenecks in the ACS Application Server on Symmetric Multi-Processing (SMP) systems can be primarily attributed to two areas: (1) Java virtual machine (VM) implementation, (2) Synchronization on the ACS Application Server's shared data structures.

Java VM Implementation. The present system utilizes a version of the Java VM wherein there is a single lock that guards the heap memory. When a thread in the virtual machine needs to allocate memory for a new object, the virtual machine acquires this lock to block other threads from allocating memory at the same time. After the VM finds a free block of memory and allocates it, then this lock is released.

An example will assist in describing scaling bottlenecks. Consider a scenario where there are a number of threads trying to allocate memory for new objects. All except one of the threads is blocked waiting to allocate memory. In terms of CPU usage, only one of the threads is utilizing the CPU while the others are not performing any useful work. This results in a scaling bottleneck.

A program might further be used to illustrate this bottleneck. For instance, the program might spawn four threads, each of which allocates an integer object forever. On a 4 CPU SMP system, this program might consume 100% of the CPU resources. However, if this program is run on a specialized system, such as Sun's JDK 1.1.7, then the CPU utilization will only be around 25%. This bottleneck is thereby circumvented by using Sun's Java2 Virtual Machine. Sun's Java2 VM with the HotSpot performance engine (for example) serves to mitigate this problem by creating thread local memory pools. Still other VMs (such as IBM's Java platform) have also improved memory allocation algorithms. Basically, these memory pools are local to a given thread. As such, threads do not have to obtain a lock while allocating memory during normal execution. When a thread runs out of memory in its local pool, it then either allocates memory from the global pool, or scavenges memory from other thread local pools.

Synchronization on Server's data structures. Server code extensively uses Java utility classes -- such as Hashtables and Vectors -- for data structures that are accessed and modified by a single thread in the ACS Application Server. Most of the methods in these utility classes are synchronized between the various servers. This is required, given the general-purpose nature of the utilities, and as such, they need to be thread safe. The cost of accessing synchronized information is high, even when there are no contentions on the monitor that synchronizes access to the method.

To alleviate this bottleneck, Java2 provides a Collections framework with additional utilities such as Hashmaps. Hashmaps provide functionality similar to Hashtables, but with much better performance characteristics for single-threaded access. Such performance critical data structures can be used for in-memory representation of XML documents, and the like.

Scaling out on clusters. Loosely coupled clusters of nodes dramatically reduces the problem of synchronizing access to hardware resources, such as memory and CPUs, which is the cause of most bottlenecks on SMP machines. Clusters, however, add the overhead of sharing data among the loosely coupled nodes. If applications running in a loosely coupled cluster do not share any data, then (in theory) the total work performed by the cluster will increase linearly as nodes are added (that is, provided until the upper limit on some shared resource such as network bandwidth is reached). Conversely, if the applications running in the cluster require a significant amount of shared data, then the shared data might be the scaling bottleneck.

As a result, the solution to scalability in a cluster is to minimize the volume of data exchanged between nodes of the cluster. Towards this goal, the ACS Application Server framework has added the concept of Logical References (see further details below). In short, a logical reference to a shared object is an identifier that can be used to re-create an object. Logical references tend to be much smaller in size than the objects that they identify. Thus, but exchanging logical references for data exchange instead of actual objects, the scalability can be improved in a loosely coupled cluster.

Availability. It is important for the ACS Application Server, and its associated services, to be available on a continual around-the-clock basis to customers and end users. This of course would apply to any other type of server providing critical services to customers and end users. The need for a transparent fail-over mechanism becomes essential in order to facilitate certain availability and zero downtime requirements. The present system envisages that a cluster of nodes would be available to take care of concurrent user requests. When one of the nodes happens to go down -- due to any of a variety of reasons, either planned or unplanned -- one of the other existing and running nodes would be able to continue the servicing of the original request that emanated from the browser to the failed node, without any apparent or discernable ill effects as far as the end user session is concerned.

Various commercial application servers (i.e., NAS 4.0 and Weblogic 4.5) support the concept of clusters, where multiple separate nodes can be configured to work in a synergistic manner. These multiple nodes in the cluster service requests from the clients and the web server can be configured to perform load balancing between the different nodes on a round-robin (or other) basis. Once a session has been initiated with a particular node in the cluster, the web server ensures that subsequent requests in that same session get routed to the node from where the session originated. This "sticky" behavior of the web server allows the present system to maintain and retrieve session state information in the application itself instead of either sending the state to the browser or adopting a stateless model.

Referring now to Figure 5, a representative block diagram 500 is shown of a primary node 502 and backup node 504 in association with a web server 506. A browser (as representative of a user or client) 508 invokes a particular application and sends an application request 520 to the web server 510 (which might operate via a proxy 512). The Web Server 510 then passes the request on to the appropriate Application Server. For instance, the primary node 502 -- which might be running the ACS application server -- might service the application request 520 from the browser 508. The Web Server 510 knows about the presence of the multiple nodes, wherein "n" number of nodes (n = 2 in this case) forms a cluster. An order entry application, for example, will result in a request by the user to display product information (or the like). The user will then select a particular item on the list and place an order. The next screen might show the items that have been selected, and the associated quantities. During each interaction with the web server, the state of the application is stored via state variables.

The primary node 502 therefore "puts" the state variables representing the application in a Session Object 530 (i.e. HTTP Session Object, or the like). These actions are generally indicated by the unbroken lines 522 and 524 respectively. The broken lines 526 and 528 represent subsequent requests, sent during the same session, which are sent upon fail-over to the backup node 504 in the cluster. Note that there can be multiple Browsers (shown collectively as 509), and each Browser can have a separate Session Object (shown collectively as 531). Hence, the Session Object -- which is relevant to the particular Browser that has opened up a session -- is where the state information regarding the status of an application pertaining to that session is stored. As described above, the Session Object might have canonical identifiers, raw data, or a combination of the two. Thereafter, the response 521 is shown sending information back to the browser 508.

Accordingly, each time the Browser makes a request, it goes from the Web Server to the Application Server. The Application Server would then send back a response, which goes back to the Browser. Each time the Applications Server sends back a response, the Session Object is replicated across to the backup node 504. Two example types of replication include (1) Replication through "the wire"; and (2) Replication through a database. Note that "the wire" might refer to any connection between the primary and the backup server, and would typically include the network connections between the two servers. This is the preferred method, as data will be continually sent to the backup server, and a fast and efficient connection is needed. Replication through a database 532 might also be used to pass such information. Interacting with a database, however, will not generally be an optimum approach, as such usage carries the added performance cost of continually accessing the database. With either approach, the backup node accesses state variables from the session object 530 and continues processing the application requests.

In the ACS Application Server, the state information is maintained at several levels. The session as a whole has a global pool, and there are several local pools associated with the currently running composite step and the stack of composite steps that the session may have processed. In addition, the ACS Application Server maintains several Object Caches on behalf of the different applications to optimize on fetches from any particular data source. The Object Pools and Object caches are node specific, i.e., when multiple nodes exist in a cluster, each of the nodes maintains its own private views of these in-memory storage structures.

When one node fails, and the request is redirected to one of the remaining nodes in the cluster, this stored session state information and Object Cache state should be available on the servicing node. This is accomplished through use of a replication feature on the cluster.

It is also necessary to facilitate a transparent and seamless fail-over. This task is accomplished using a messaging interface.

Replication. A variety of commercial application servers support the replication of information between nodes in a cluster via the HttpSession Object (or Session Object). In particular, this would generally include replication between the primary node and the backup node. The Session Object is created when the first request from the browser is issued to a Servlet and is maintained by the Application Server. This Session Object also serves as a "container" for storing and retrieving state information in the form of an name value pair using the putValue( ) and getValue( ) API (Application Interface) that is available in the JDSK (Java Servlet Development Kit). If the object that is being "put" into the Session Object is an instance of a class, then it should implement the Serializable Interface so that a Serialized representation of the instance is stored in the Session Object.

In a clustered environment, replication of the Session Object between the primary and the designated backup nodes is effected using RMI. RMI (Remote Method Invocation) is a way that a programmer, using the Java programming language and development environment, can write object-oriented programming in which an object on different computers can interact in a distributed network. RMI is the Java version of what is generally known as a remote procedure call, but with the ability to pass one or more objects along with the request. The object can include information that will change the service that is performed in the remote computer. Sun Microsystems calls this "moving behavior." For example, when a user at a remote computer fills out an expense account, the Java program interacting with the user could communicate, using RMI, with a Java program in another computer that always had the latest policy about the expense reporting. In reply, that program would send back an object and associated method information that would enable the remote computer program to screen the user's expense account data in a way that was consistent with the latest policy. The user and the company both would save time by catching mistakes early. Whenever the company policy is changed, it would require a change to a program in only one computer.

The present system can effectively leverage the replication characteristics of the Application Server and replicate the session state information and the local pool information that needs to be maintained by an application "putting" this state information into the Session

Object, just prior to the application sending the response back to the browser. If (and when) the primary node fails and the session is redirected to the designated secondary node, replication of the Session Object would occur and the session state information retrieved from the Session Object to ensure a seamless fail-over. The fact that the incoming session's session ID information is not present in the Internal Session Table of the current application would enable one to deduce that fail-over has occurred. Further since the system "put" the IP Address of the originating node in the Session Object, the present system can verify that the incoming session is a failed-over session, and not a "timed out" session. A timed out session would also cause an absence of the session ID in the internal Session Table. Once a session is deemed to be failed-over session, the relevant state information can be reconstructed from the values stored in the Session Object with deserialization as necessary.

When a secondary node services a failed-over request, the node "puts" its IP address in the Session Object. This node then effectively becomes the primary node for the session. Thereafter, the session appears as if it originated from this node.

Object serialization and deserialization can be expensive processes. The ACS Application Server needs to maintain objects that are often large and cumbersome to serialize/deserialize using the Java Object Serialization Interface (or the like). Accordingly, the present invention employs the concept of Logical References. A Logical Reference essentially defines the reconstruction behavior of any object that needs to be stored as a Session State. Every object that is put into any of the Object Pools needs to implement the Logical Reference interface, which is basically an optimized serialized/deserialized interface.

Object Cache synchronization. While replication solves the problem of Session State synchronization, it does not resolve problems relating to Object Cache synchronization. Object Caches could potentially be large, and it would be impractical to replicate them. Moreover, when specific objects in a cache are invalidated, these actions need to be performed on all caches across the nodes in the cluster.

A suitable Publish/Subscribe API can be utilized to post cache invalidation messages from one node in the cluster to all of the rest. Figure 6 shows certain representative elements 600 to demonstrate this process. A request from an Application/browser 602 is shown coming into a Web Server & proxy 604. The primary ACS Application Server 606 is shown to have an associated Publisher/Subscriber API 608, and an Object Cache 610. A secondary ACS Application Server 612 is shown to have its own associated Publisher/Subscriber API 614, and an Object Cache 616. The ACS Application Server, which runs the subscriber thread, receives these messages and invokes the appropriate message processor to process the message. An interface ACSMessage (or the like) is defined to process the message and the Object Cache class implements the ACSMessage interface. When an Object Cache is instantiated by an application, it registers the cache type with the Subscriber so that the appropriate message processor can be invoked on receipt of a message based on the cache type of the incoming message.

The transport protocol for the Publish/Subscribe Interface can be based on the JMS API, making use of the Application Server and the JMS Server and JNDI repository, and creating a named Message Topic therein. Alternatively, the Java Multicast API can be utilized to create a multicast socket at a predetermined multicast address and port (620). The Publishers would then send the cache invalidation messages 618 to this port 620. Subscribers would wait on this socket as well and receive messages, which would be processed by the appropriate message processor. The present implementation uses the Java multicast API for the publish/subscribe interface.

The present system for providing availability is further defined in terms of examples. Referring to Figure 7A, the block diagram shows that each Application 702 will have associated with it a defined state 704. Figure 7B shows an example of the Browser 710 invoking an application 712, such as "Search For Product." In running this search routine, a query 714 will be used. The result might produce a number of items 716 (i.e., 40 items). The display can be selectably configured to show only 10 items per page (718). As the user moves from page to page, this number is tracked. For instance, this example state includes the user being on page 2 (720). The defined state 722 for this particular application then becomes the query (714), the page size (718), and the page number (720). This prevents having to serialize and send over the content relating to all 40 items that resulted from the query. With these few pieces of information, the search query can be reconstructed -- in the event of fail-over -- and the same resulting items and relative page numbers can be displayed to the user.

The definition of a particular state must be created for each application. Each time the user interacts with the application, the state changes. The efficiency of keeping track of the state is therefore important. One brute force method for keep tracking of state information is to remember (via a database or otherwise) all the requests, and then replay them on a backup server. This is not very efficient, given that thousands of tasks might be queued up on a first machine, and it would not be desirable to process those same thousands of tasks on a second machine in the event of a failure on the first machine. In contrast, however, by sending over the entire set of objects, no re-construction of those objects would need to be performed. This trade-off must be addressed.

In order to make the process of transferring state information more efficient, the above-mentioned concept of Logical References is further used. Instead of sending over entire sets of objects, only a key portion of the data is sent over, wherein this key information can be used for full object reconstruction. Note that object reconstruction itself might take considerable time and processing power. This tradeoff is reasonable, however, given that this key portion of the data will be sent over to the backup server continuously. A smaller amount of data will result in a more efficient send-over. Hence, if an object had (for example) 1000 bytes of data, then only 1/10th, or even 1/100th might be extracted as "keys" and sent over "the wire" in the form of logical references. The logical reference serves as a canonical representation of the full object. Note that "the wire" might refer to any connection between the primary and the backup server, and would typical include the various network connections (Internet, WAN, LAN, or otherwise) between the two servers.

Yet another example of the benefit of using Logical References (or keys), database records can be quickly referenced. For instance, an employee record might be stored in a database. This record in its entirety might include the employee number, employee name, job, status, manager name, and so forth. The entire record might be very large in terms of overall size. One solution, however, is to send only a key to the object, which might be a single number, symbol, or the like. If/when there is a failure, the backup server can take these various keys and retrieve the information from the database in order to reconstruct the objects. This approach readily provides a transparent fail-over, despite certain delays that might be associated with recovering the object data (via the keys) and reconstructing the objects.

Note that certain objects may not have a key by which to conveniently reference them. In such instances, a combination of both the keys and the full (or partial) data objects might be sent over the wire. For example, when dealing with a shopping cart, a customer might place an order by going into a database and viewing the products that are stored there, wherein each product will be identified by a unique key. The customer might order 10 units of product A, 20 units of product B, and 30 units of product C. While the products themselves are in the database, the number of units desired by the customer are not. Hence, in the event of fail-over for this shopping cart application, the units (10, 20, and 30) need to be sent over, as well as the product identifiers. A combination of information is sent over including the data that user has entered, along with canonical portions of the data that are already in the database. From this set of information, the shopping cart order can be reconstructed.

Note that a Session Object might need to become very complicated in order to store the various data relating to the reconstruction of objects. The backup server also needs to parse through this Session Object in an efficient manner in order to reconstruct the various objects. Accordingly, the present system utilizes Java Pools in order to organize the Session Object data. Figure 8 shows a representative hierarchy 800 of the pool structure. A Global Pool 802 is shown to include a Composite Step Pool 804, an Interactive Step Pool 806, and an Application Step Pool 808. The Composite Step Pool is further associated with a Simple Step Pool 810, wherein composite steps are comprised of a series of simple steps. The Session Object 820 is then shown to store a series of states 822 (i.e., 1 through N) for the Composite Step Pool 804 (and Simple Step Pool 810). The Session Object 820 can also store a series of states 824 (1 through N) for the Interactive Step Pool 806. As a further example, the values for the Application Step Pool are shown stored as a Java Hash Table 826. For this example application, the state variables 828 are defined to include an integer "i" and a string "s". In State 1, i = 10, and s = "abc"; in State 2, i = 10, and s = "def". For a particular state (i.e., State 1), the Session Object 820 stores the Hash Table values i = 10 and s = "abc". The Hash Table approach facilitates efficient lookup of the various Logical Reference (or raw) data within the Session Object. The Pools are therefore gathered in place in the Session Object.

Note that other approaches might employ session pools, or the like, in order to store data that might be used to reconstruct the state. However, the present invention adds the ability to form a canonical representation of the state that needs to be stored, replicating the data periodically over to the backup node, and then reconstructing the state based upon this canonical representation that is periodically replicated. For objects in a database, a key can be used as the canonical representation. For a page, the canonical representation might include the query, page size, and current page position. Hence, each object in the system has its own Logical Reference, and the Logical Reference for a business object is an identifier for that object. The canonical representation varies with the kind of object that is being represented. In general, this provides for a more efficient system.

Figure 9 refers to yet another example of a Keyword Search Application in relation to a representative cluster structure 900 (as similar to Figure 5). A Browser 902 is shown interacting with the Web Server (and Proxy) 904. For a cataloging situation, the first page might appear to be a page 903 that lists certain products and their descriptions. This HTML page might include an option to perform a keyword search. Once entered by the user, an application request 906 in the form of a Keyword Search 908 is shown. The keyword in this particular example is "Router" 910. The request 906 is forwarded from the Web Server onto the Primary Node 912. As the particular Session (as associated with the particular Browser) progresses, state information pertaining to this search will be placed in the Session Object 914. The Session Object 914 will contain Serialized State Information 916, which includes Logical References (i.e. keys) and/or full data (for objects without keys). The Response 930 coming back to the Browser again displays the result on the relevant page 903. Options are provided to select "Next Page" 932 and "Prev. Page" 934, wherein a new request will be sent back to the Web Server 904.

This particular application is an example of a Composite Step 1000 (see Figure 10), with the entry point being a search step, and one of the parameters passed into the search term being "Router." Step 1002 next shows the formulation of a query associated with this search term, wherein all the attributes of the product object will be searched for a product description such as "Router." Step 1004 shows the execution of this query. Step 1006 shows the process of getting the results back from the database, which might provide several rows of product objects that match this particular criteria. Based upon this, the application will next construct a vector of product business objects which matches the user specified condition. This vector will be stored in a "page iterator," which has a vector of business objects, a selected page size, and a current page position. Once this information is formulated, the Composite Step Pool 1007 is formed, which will include the page iterator, an integer representing the page size, and an integer representing the current page position. As shown in this example, the page iterator might be 1 or 2, the page size is 5, and the current position might be 1 or 2. Step 1008 next shows the serialization of the pool information using Logical References (where appropriate). A string or single select statement might be used to represent the page iterator, while an integer can be used to represent the page size and current page position. In step 1010, the serialized data is thereafter stored in the session object.

For any further interactions by the user with the page, the state will likely change. For instance, if Next Page (932) is selected, the current page position will be incremented by one. This new information (along with the prior information regarding page iterator and page size) are serialized into a new state and stored in the Session Object under the appropriate pool.

Note that while a Composite Step is described, the functionality of the general steps applies to other step types as well. As shown in element 1100 of Figure 11, the step type might include Composite Steps (with Simple Steps), Interactive Steps, and Applications Steps. In progressing through the workflow, 1102 shows the execution of the step. The pool information is next determined in 1104. The pool information is then serialized with Logical References (where appropriate) in 1106. This serialized information is then stored in the session object, as shown in 1108.

Thereafter, if the Primary Node (Application Server) (912) is down, then the Web Server 904 will next go to the Backup Node (Application Server) (918). From the Session Object 914, the Backup can parse out the relevant Logical References to the object information, and reconstruct the contents of the desired page (i.e., current page number).

Note also that with the ACS server configuration, there are many different applications built on top of the platform, all of which run on the same device. Under the presently described configuration, the various applications do not need to be written to account for fail-over situations. Instead, the server will take care of fail-over in a transparent manner via redirection to a new Application Server through the Web Server.

Manageability. Manageability can include many features, including but not limited to adding/dropping cluster nodes; starting and stopping application servers; deploying applications in a cluster; enabling/disabling specific applications in a cluster; and cache administration.

Adding and dropping of cluster nodes. This feature can generally be managed entirely by the commercial application server. In general, an administrator will need to insure that each new cluster shares the same general installation as all of the other members. A node configuration file will need to be created or updated as necessary. This step is generally not necessary if all of the nodes in the cluster are uniform. For every web server in the configuration, the web server will need to be disabled in the load balancer configuration, the web server will need to be shut down, and the web server configuration file will need to be edited to include the new node. Thereafter, each web server should be restarted, and then enabled in the load balancer configuration. Once all the web servers have been reconfigured, the application server should be started on the new node. The server will automatically join the cluster when it boots up, and will be ready to service requests thereafter. The node should next be added to the ACS Application Server cluster configuration file.

To permanently drop a node from the cluster, administrators need to edit the web server configuration and the ACS cluster configuration to remove the node from the list of nodes in the cluster. This generally involves, for every web server in the configuration, the steps: disabling the web server from the load balancer configuration; editing the object configuration file; removing the node port reference; reloading the configuration files; restarting the web server; and enabling the web server in the load balancer configuration. Thereafter, the ACS Application Server cluster configuration file should be edited to remove the node from the list of nodes in the cluster.

Starting and Stopping the Applications Servers. Most commercial application servers (like Weblogic) provide a console for starting and stopping the servers. They generally require an administrator password. The user then identifies the node to be started or shut down. A command is entered accordingly to accomplish the result.

Deploying Applications in a cluster. A cluster tag in a Server.xml file is generally used to indicate the fact that the Server is running in a clustered configuration. A value of "0" might be used to indicate a standalone configuration, while a value of "1" indicates a clustered configuration. Servers that are clustered are started up as servlets with the appropriate properties, and in particular specifying the replication type to be in-memory. At the time of the application server startup, the multicast address used for the aforementioned "heartbeat" detection communication between nodes in a cluster is also specified.

In addition to the multicast address used for heartbeat detection by the proxy, an ACS cluster also needs to communicate between individual nodes in the cluster to send and receive messages. The transport for this messaging interface is the multicast API. The Server.xml file might therefore be used to define two additional tags when a cluster is employed. A <multicast Address> tag indicates the multicast address that will be used by all nodes in the cluster, and <multicast Port> defines the port at which all nodes will be sending/receiving messages. The multicast group can be specified by: a class D IP address, which is in the range 224.0.0.1 to 239.255.255.255; an inclusive multicast Address; and by a standard UDP port number (multicast Port). Moreover, a variety of proxy mechanisms can be used to communicate between the web server and the application server nodes in the cluster.

Once the ACS Application Servers are properly setup and configured to operate in a clustered environment, applications can define the state specific objects that need to be saved in any of the object pools using the Logical Reference Interface. Applications would continue to save state information as they would in a non-clustered environment and would be unaware of any clustering effects until a failure happens when the failed session would transparently be redirected to the backup node in the cluster for further processing. Logical references are necessary to obviate the usage of the default Java Object Serialization API, which can have adverse effects on performance throughput.

Applications that create and maintain Object Caches in the Server using the Object Cache Manager interfaces would continue to work as before. It is the responsibility of the Object Cache Manager to ensure that appropriate cache invalidation messages are published to all nodes in a cluster when running in a clustered configuration. Applications need to define a specific cache type in the ACS Message interface and use this cache type during cache instantiation. This would facilitate the Subscriber to invoke the appropriate message processor on receipt of a cache message for further processing.

Enabling/Disabling Specific Applications. Enabling and disabling of the applications can be done at run time through the Administration Tab of the ACS service. An application can be disabled immediately in which case the user would not be able to perform any other action on that application, or it could be disabled gracefully in which case the current users would be able to finish what they are doing, but no new user would be allowed to use the application. Once all the current users are logged out or their sessions expire then the application would be disabled. When the applications are enabled or disabled at run time the changes are made persistent by writing them to the database.

The other nodes in the cluster are notified of the changes through the messaging infrastructure. For this release an application is enabled and disabled across all nodes in the cluster.

When the system is running, the user should only be able to use an application if it is enabled. When the ACS Application Server is initiated, it gets the information about the application status (enable/disable) from the application configuration files. If an application is disabled, an indicator such as a grayed-out icon might be used. If the user jumps from another application, then a disable application error message would display the fact that the application is not available for use.

Cache Administration. Administration of caches can be done at runtime through the Administration Tab of the ACS service by going to the cache administration section. If the service is running is a clustered configuration, then the caches could be administered across the cluster, or on a node by node basis. Some properties of the caches can only be administered across the cluster while others could be administered on a node by node basis. Clearing and reloading of caches are done at the cluster level, while aspects such as getting the statistics, or getting/setting of cache parameters like the capacity are done on a node by node basis. The settings required for the cache administration are specified in the Server.xml file. All the settings that apply to all the nodes in the cluster are specified in an associated Configuration.xml or Server.xml file. The node specific settings are specified in node name file. The fact that the system is running in a cluster is specified by the <cluster> tag if set to "1". In a cluster configuration, the names of the nodes in the cluster are specified within a <Cluster Nodes> tag. The nodes in the cluster could either use the cluster wide settings or overwrite them by specifying them within the node settings section. Cluster wide settings, as indicated by the appropriate tag, contain all the parameters, while the node specific settings indicated by a <Node Setting> tag only contain the parameters that they overwrite.

Although the foregoing invention has been described in some detail for purposes of clarity of understanding, it will be apparent that certain changes and modifications may be practiced within the scope of the appended claims. Therefore, the described embodiments should be taken as illustrative and not restrictive, and the invention should not be limited to the details given herein but should be defined by the following claims and their full scope of equivalents.